Parallel and Concurrent Programming

Practical Written Work 1: Parallel Reversi

**Student ID: 30001979 / Student Name: Nathanail Kinnis**

**Module Code: CS3S666**

**Module Team: Marius Miknis, Gauis Mulley**

**Date Set: 24-Sep-2021 23:55**

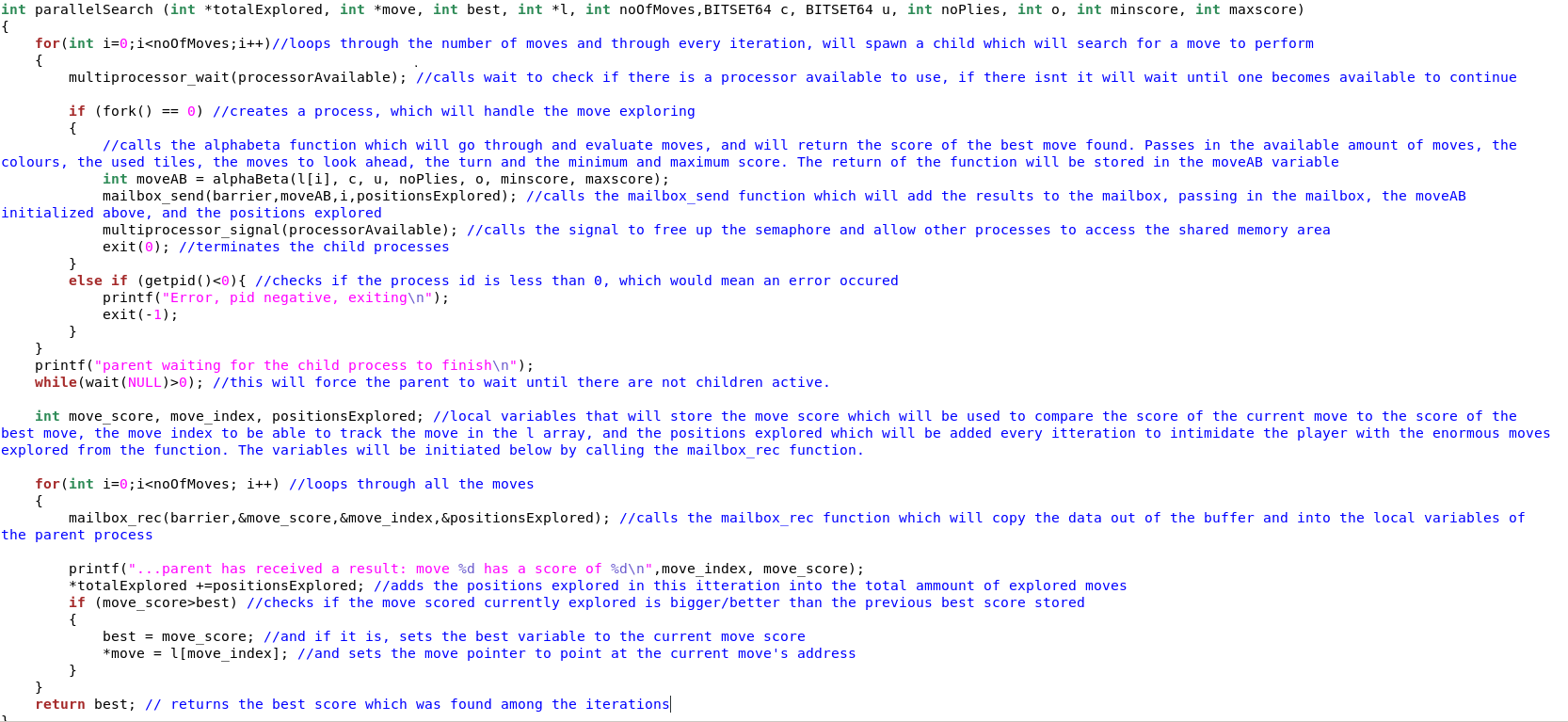
**Submission Date: 12-Nov-2021 23:55**

**Return Date: 10-Dec-2021 23:55**

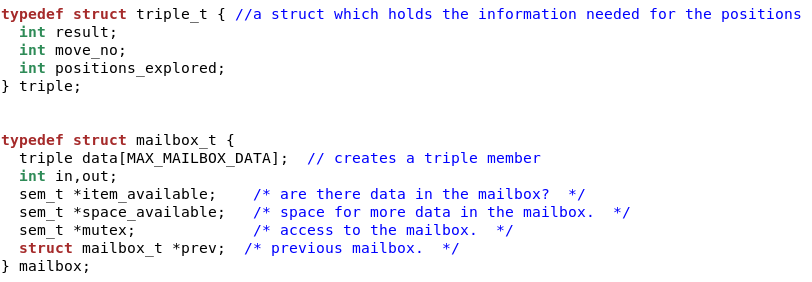
**Analysis and future improvements**

**Analysis**

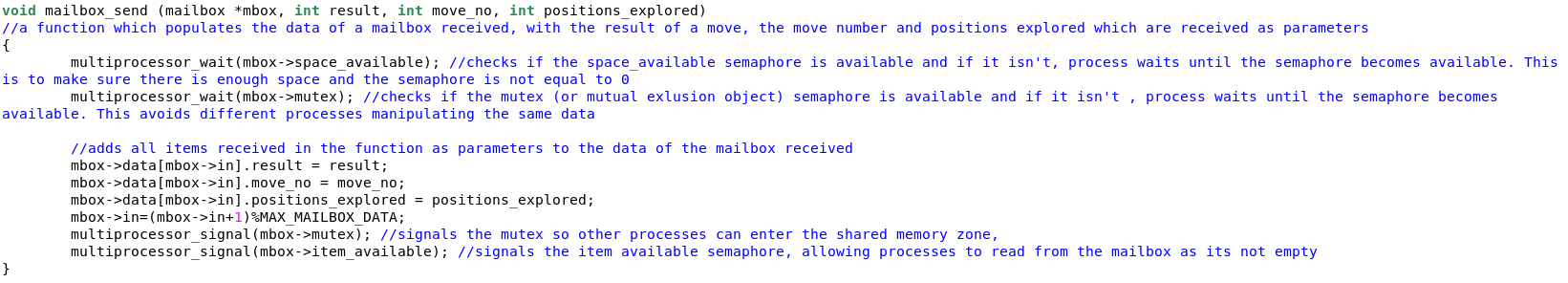
Due to the high volume of moves which the 'computer player' has to analyse, sequential searches and usage of a single processor is not enough as the time the processing of a move takes is enormous. To allow the game to reach its full potential and calculate as many moves as possible in a relatively small period of time, the introduction of parallelism was made, utilizing the most out of the processors available on each machine. The parallelSearch function will scan through available moves and evaluate each one, and once done, return the best move found its caller

parallelSearch function in paro64bit.c 

The function above loops through all available moves, and for each iteration if there is an available processor, spawns a new child process, which will scan one of the moves available while tree searching a number of moves ahead, evaluating the quality of that move and setting a score of quality to that move. Each of the moves stores the relevant data using the struct called mailbox.t through the function mailbox\_send shown below, which is called from the parallelSearch Function shown above:

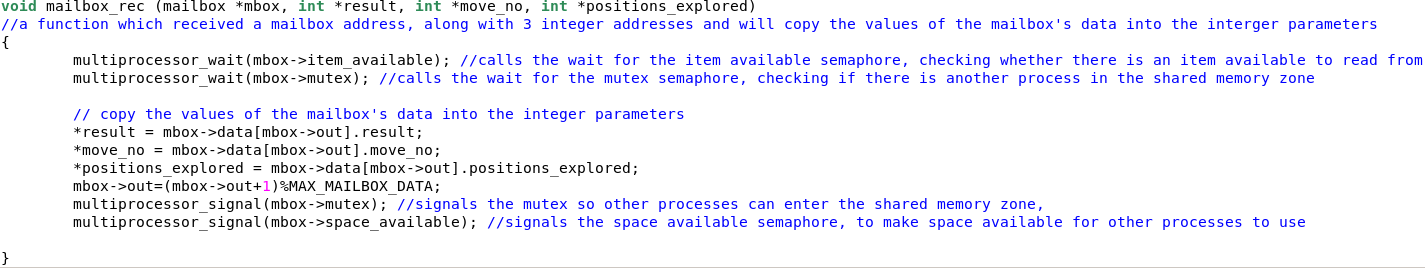
mailbox.t and triple.t structs from mailbox.h code:

mailbox\_send function from mailbox.c code:



Once all move trees have been searched, all scores will be compared and the move with the highest score will be returned from the function and played by the computer player. This is done through the parent process in the parallel search function, acquiring the need data through the function mailbox\_rec function.

mailbox\_rec function from mailbox.c code:



**Future Improvements:**

The time search of each move may vary and there is no way of predicting the amount of time for a process to cover down a tree, with multiple variables being included in how much processing a search requires. To make the searching of the move trees more efficient, whenever all moves have been assigned to a processor, if any processors are left unused, assign them to assist on existing searches of trees, splitting the work in two. Furthermore, the above can also be included when a processor finishes a search they were assigned while other searches are still processing, with the finished processor being reassigned to a new existing search.

In addition, the writing of code could be improved as the current form of code is hard to read. Renaming variables to more self-explanatory names instead of just the initials and including documentation/comments could significantly revamp the readability of the code.

Lastly, the inputs received can be inconsistent at times, by either accepting invalid options with no number (such as aa), or not accepting valid ones as shown in the images below. Adding further checks to the makeMove function would eliminate the inconsistensis

Image of not accepting a valid input: Image of accepting an invalid input:

Shape, arrow

Description automatically generatedText

Description automatically generated

**Full code**

**Due to the large space the code took, I uploaded all relevant code to a github repository**

**Link:** [lordexishigh/Parallel-and-Concurrent: This is for the Parallel and Concurrent assessment. Student ID: 30001979 (github.com)](https://github.com/lordexishigh/Parallel-and-Concurrent)